Improving Code Generation for Software-based Error Detection

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden, Dresden, Germany
Center for Advancing Electronics Dresden (cfaed)
Email: {firstname.lastname}@tu-dresden.de

***Abstract*—With modern hardware heading for smaller feature sizes and lower operating voltages, failure rates are expected to increase. Instead of adding hardware to protect against failures, which is expensive and inflexible, one can implement error detection in software. The focus of the present work is the *AN encoding* scheme. Improvements to code generation are introduced that enable a high level of fault coverage, namely over 98%, while reducing the performance overhead due to AN encoding by up to 45%. This raises the general question of whether code generation strategies can be designed that enable fault detection with a minimal impact on performance.**

I. Introduction

In an effort to maintain the exponential performance growth that hardware has been experiencing for the last decades, modern hardware is heading for smaller feature sizes and lower operating voltages. This leads to reduced reliability in both processors [1] and memories [2]. Furthermore, operating *dark silicon* at near-threshold voltage [3] will render computations unreliable. At the same time, the omnipresence of embedded devices in every-day life and industry calls for a high level of reliability, especially in safety-critical applications.

In the future, more resources will have to be spent on protecting against hardware faults. Hardware-based protection is expensive and inflexible. Software-based fault detection and recovery [4], on the other hand, is cheap and can be easily adapted to changing fault scenarios. In a prominent software-based approach the values and operations of a program are *encoded*, and faults are detected by checking whether values are valid code words. To facilitate encoding and checks, extra instructions have to be added to the program, which significantly extends execution time. Longer execution times are justified if unreliable results are turned into reliable ones.

This paper studies the *AN encoding* scheme [5]–[7]. It is shown how the placement of instructions for checking leads to better *fault coverage*, i.e. a higher percentage of detected faults. To this end, we make two improvements to code generation: (i) *improved checks* and (ii) *pinning of checks*. Remarkably, using improved checks also reduces the performance impact of AN encoding.

The structure of this paper is as follows. The concept of AN encoding is introduced in Section II, while Section III describes our specific implementation of AN encoding. Our suggested improvements are assessed in Section IV, and our results are analyzed in Section V. Related work is discussed in Section VI. Section VII concludes our paper.

II. Background and Motivation

Fault detection schemes rely on redundant representations of data: AN encoding uses additional bits to represent encoded values. To encode data, a 32-bit constant A is fixed, and any 32-bit integer value n is replaced with its encoded version $\hat{n} = n \cdot A$. In order to avoid overflow due to encoding, $\hat{n}$ must be represented by 64 bits. In AN encoding, a *check* consists of evaluating the boolean expression

$$\hat{n} \bmod A = 0. \tag{1}$$

Whenever this evaluates to `False`, a fault is detected. Note that this check requires an expensive modulo operation.

AN encoding is straightforwardly extended to pointers by regarding the address stored in a pointer variable as an integer: an address-valued variable p is replaced with $\hat{p} = p \cdot A$. Checks on $\hat{p}$ are performed in the same way as above[1]. Note that since modern 64-bit systems have 48-bit address buses, no overflow will occur due to AN encoding provided $A < 2^{16}$. In the present work we treat encoding of pointers as an optional extension of AN encoding.

Since AN-encoded programs operate on encoded data, operators must be replaced with encoded versions too. A detailed account of how individual operators are treated can be found in [6]. Here we put particular emphasis on how vulnerabilities arise from certain encoded operations. We denote as $\hat{m}$, $\hat{n}$, $\hat{p}$ the encodings of the values m, n, p respectively. Encoding simple arithmetic operators is straightforward, e.g.

$$\hat{n} +_{enc} \hat{m} = \hat{n} + \hat{m}.$$

Note that if pointers are encoded, replacement of operators must also be applied to pointer arithmetic. To encode bitwise operators, operands must be decoded:

$$\hat{n} \,\&_{enc}\, \hat{m} = (n \,\&\, m) \cdot A.$$

Code is vulnerable whenever it operates on the non-encoded values m, n. Memory operations are equally vulnerable, even if pointers are encoded:

$$\texttt{load}_{enc}\, \hat{p} = \texttt{load}\, p.$$

To alleviate the vulnerability due to operating on non-encoded values, encoded values should be checked before decoding. For bitwise operations this leads to the dependency graph in Figure 1a; for memory operations on encoded pointers the

[1]The semantics of the chosen implementation language may require casting p to an integer before encoding it.

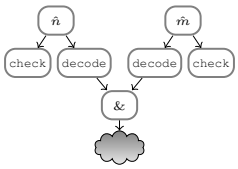

Fig. 1a: Bitwise operation with checks.

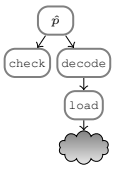

Fig. 1b: Memory operation with checks.

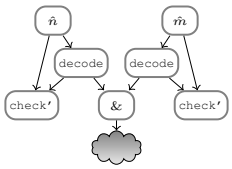

Fig. 2a: Bitwise operation with improved checks.

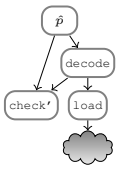

Fig. 2b: Memory operation with improved checks.

graph in Figure 1b is obtained. In both graphs a `check`-node is implemented by evaluating the boolean expression (1).

To understand our first improvement, namely improved checks, note that the `check`-nodes in Figures 1a and 1b are independent of the `decode`-nodes. Thus the compiler may schedule instructions for these nodes far apart, leaving intermediate results vulnerable to faults. Using improved checks, depicted as `check'`-nodes in Figures 2a and 2b, reduces the sizes of vulnerable code sequences: since `check'`-nodes depend on `decode`-nodes, instructions for checking will be placed more tightly around the &-operation and the `load`-instruction respectively. The `check'`-nodes are implemented by evaluating the following boolean expression:

$$n * A = \hat{n}. \tag{2}$$

This implementation also improves performance since it does not rely on an expensive modulo operation. Compilers without sophisticated strength reduction will therefore produce faster code.

Our second improvement is built on a simple observation: Independent of how checks are implemented, a compiler may be able to infer, in certain situations, that $\hat{n}$ is a multiple of A. If this is the case, and based on the optimization level, the compiler may decide to delete instructions for checks. This behavior is undesirable since the observation that $\hat{n}$ is statically a multiple of A ignores the fact that faults occur dynamically. To avoid that checks are optimized away, we can *pin* checks. This is achieved by placing a pseudo-copy instruction around the argument of a check. The result of the pseudo-copy is then fed to the actual check. The pseudo-copy instruction is replaced with a conventional move instruction immediately before register allocation. Thus checks cannot be optimized away, but the register allocator can still coalesce source and target register of the pseudo-copy. This technique was already applied in [8]. Here we analyze its impact on fault coverage and performance.

strategy	type of check	pinning
baseline	(1)	✗
pinned	(1)	✓
improved	(2)	✗
imp., pinned	(2)	✓

TABLE I: Code generation strategies.

III. Implementation

Our implementation of AN encoding is based on the encoding compiler framework that was introduced in [8]. The code transformations that facilitate AN encoding are implemented at the level of LLVM intermediate representation (IR) [9]. However, unlike in [8], only a minimal number of checks is inserted into AN-encoded programs. Specifically, there are only three places where checks are performed:

(a) on the results of a `load`-IR-instruction,
(b) on the non-pointer argument of a `store`-IR-instruction,
(c) on values that are decoded, as explained in Section II.

The checks in (a) ensure that only valid code words enter the program's data-flow. This serves to protect memories, including caches, against hardware faults. If a fault occurs during computations on encoded data, it is highly unlikely that a subsequent fault will turn the corrupted data word back into a valid code word. Therefore the checks in (b) suffice to verify that the final results of computations are valid code words before they are committed to memory. Intermediate checks are not necessary, except where values are decoded, cf. (c). When pointers are encoded, the pointer arguments to `load`- and `store`-instructions must be decoded immediately before these instructions are executed, as in Figures 1b, 2b. Checks are then performed on the pointer arguments due to (c).

Table I gives the definitions of the code generation strategies for AN encoding that are analyzed in this paper. In addition, encoding of pointers can be optionally enabled.

IV. Experimental Setup and Results

The strategies from Table I are applied to the following benchmark algorithms: **Matrix-Vector Multiplication**, **Array Copy**, **Bubblesort**, and **Quicksort**. This set of algorithms represents canonical features of computation, namely arithmetic operations, data movement, and control-flow that cannot be predicted at compile-time. The generated executables are subjected to fault injection experiments and performance measurements, which are conducted on an Intel Core i7 CPU running at 3.6GHz with 32GB RAM.

For fault injection we use Intel's Pin tool [10] together with the BFI plug-in[2]. A single fault is injected into a given test program at run-time as follows. First, one of the instructions executed by the program is chosen at random. Then, a single or multiple random bits are flipped in one of the registers manipulated by the instruction. This fault injection procedure is suitable for simulating transient faults in the combinational logic of a CPU since such a fault will manifest itself in a wrong result being stored in a register.

[2]https://bitbucket.org/db7/bfi

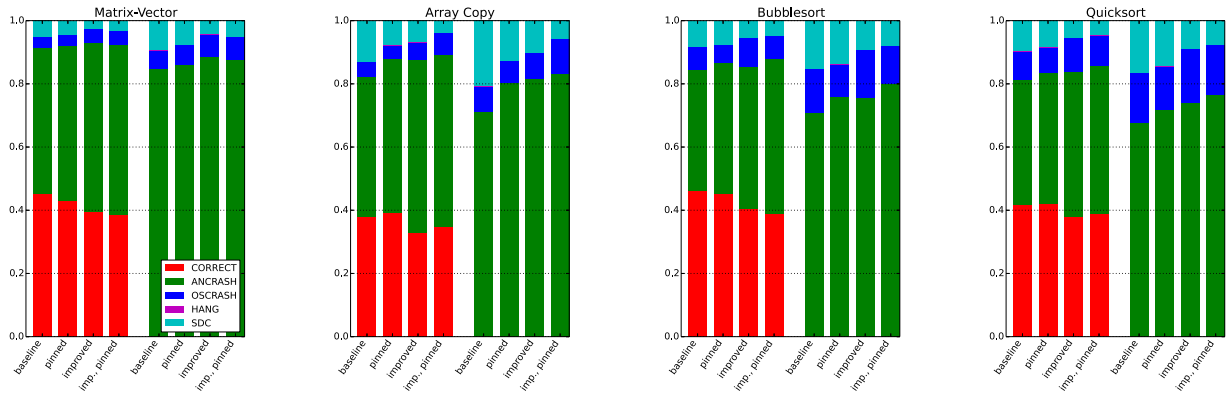

Fig. 3: Fault coverage for encoding of integers only.

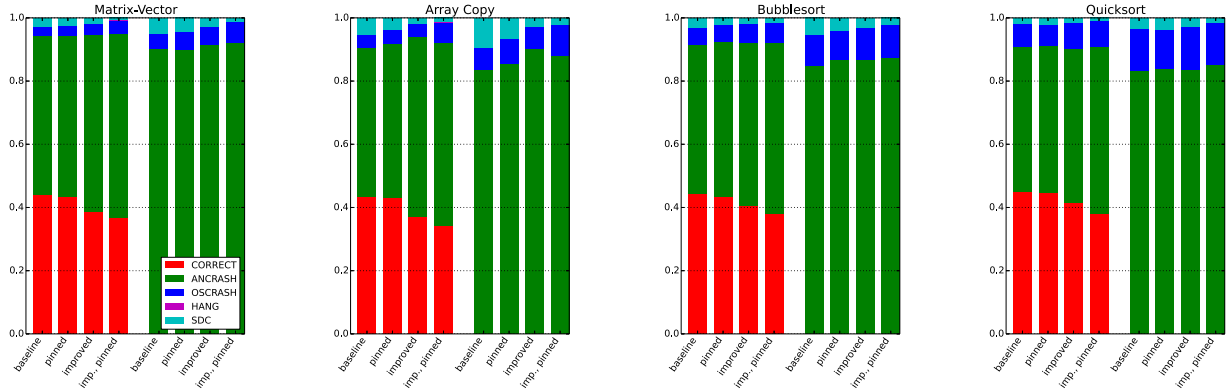

Fig. 4: Fault coverage for encoding of integers and pointers.

For each of our benchmark algorithms combined with each of the code generation strategies we repeat the fault injection procedure 10,000 times. In each of the fault injection experiments program behavior is classified into one of five categories:

1) *CORRECT*: Despite the injected fault the program produced correct output and terminated normally.
2) *ANCRASH*: The injected fault has been detected by one of the inserted checks.
3) *OSCRASH*: The injected fault caused the operating system to terminate the program, e.g. due to a segmentation fault.
4) *HANG*: The injected fault caused the program to take more than $10x$ its usual execution time. The program is therefore deemed to hang.
5) *SDC*: Silent data corruption has occurred, i.e. the program terminated normally but produced incorrect output.

Software-based error detection aims to reduce the frequency of SDC. For the purpose of evaluating our results we therefore formally define *fault coverage* as the frequency of non-SDC results after fault injection. Figures 3 and 4 show the frequencies with which events from the five categories occur. Each of the bars corresponds to one of the strategies from Table I. The sets of bars on the right of each plot are based on the same data as the bars on the left, only the CORRECT events have been left out. We will discuss our findings in detail in Section V.

To measure the performance impact of different code generation strategies, we follow [11] to obtain cycles counts. The number of cycles taken by each executable generated with one of our strategies is divided by the cycles taken when no AN encoding is applied. The resulting quotient is reported as the slow-down due to AN encoding in Table II. The numbers in Table II were obtained for an input array size of 1,000 elements, where each element is a 64-bit word.

V. Discussion of Results

Figures 3 and 4 clearly show that pinning and improving checks both lead to increased fault coverage. If applied individually, improving checks is more beneficial to fault coverage than pinning checks. The highest coverage is generally achieved when both strategies are combined. It is interesting to note that more aggressive checking reduces the CORRECT proportion. This is because faults that do not adversely affect program execution are more likely to be detected if many checks are performed.

	integer encoding only				*integer and pointer encoding*			
	baseline	improved	pinned	imp., pinned	baseline	improved	pinned	imp., pinned
Matrix-Vector	15.72	15.34	18.43	14.12	24.70	23.69	32.29	22.18
Array Copy	7.50	6.07	12.08	8.90	22.30	17.66	29.38	24.63
Bubblesort	3.46	2.77	3.93	3.18	8.08	6.31	9.52	7.21
Quicksort	1.95	1.76	1.98	1.79	3.42	3.00	3.92	3.22

TABLE II: Slow-down due to AN encoding.

	integer encoding only		*integer and pointer encoding*	
	improved over baseline	imp., pinned over pinned	improved over baseline	imp., pinned over pinned
Matrix-Vector	2.5%	30.5%	4.3%	45.6%
Array Copy	23.6%	35.7%	26.3%	19.3%
Bubblesort	24.9%	23.6%	28.1%	32.0%
Quicksort	10.8%	10.6%	14.0%	21.7%

TABLE III: Speed-up due to improved checks.

Our code generation strategies do not only increase overall fault coverage but also the proportion of detected faults, i.e. the proportion of ANCRASH. The single major exception to this rule is the Array Copy benchmark in Figure 4: when improved checks and pinning are combined, the proportion of OSCRASH increases and takes away some portion of ANCRASH. This behavior is caused by faults that affect the program counter, which are responsible for the vast majority of OSCRASH events in the Array Copy benchmark. When improved checks are also pinned, more checks appear in the generated code. Each check is accompanied by a conditional jump instruction to code that should be executed if the check fails. The only register that is modified by the jump instruction is the program counter. This means that more checks in the Array Copy benchmark imply that more faults are injected into the program counter, which, in turn, leads to a greater proportion of OSCRASH.

Figure 3 can be summarized by noting that fault coverage is raised from a low of 87.1% in the baseline strategy to above 95.2% when checks are improved and pinned. When pointers are also encoded, i.e. in Figure 4, coverage is raised from a low of 94.6% to above 98.6%. Protecting pointers is particularly relevant to the observed fault coverage since all of the benchmarks operate on arrays: if a fault causes a bit-flip in the lower bits of an address, it is very likely that the corrupted address is still within the range of the array. Reading from the corrupted address will thus return a valid code word, and hence the fault cannot be detected. However, the computed result will still be incorrect. The effectiveness of encoding pointers comes at a price, as can be seen from Table II. When pointers are encoded, even the slow-downs for the baseline strategy are significantly worse than any of the slow-downs for encoding integers only. The reason for this is that encoded pointers require that every memory access is accompanied by an expensive division operation.

Table II shows that if only integers are encoded, the Matrix-Vector Multiplication benchmark incurs by far the greatest slow-down. This is due to the expensive encoded version of multiplication, cf. [6], [8]. When pointers are also encoded, the slow-down of the Array Copy benchmark is similar to that of Matrix-Vector Multiplication. In other words, the Array Copy benchmark is the one that suffers the worst from pointer encoding. This is unsurprising given that the Array Copy algorithm essentially consists of memory accesses. Table II also proves our claim from Section I that improved checks reduce the slow-down due to AN encoding. For definiteness the speed-ups achieved by using our improved checks are listed in Table III. The best speed-up, namely 45.6%, occurs for the Matrix-Vector Multiplication benchmark.

We conclude this discussion by comparing with previously reported results. The levels of fault coverage achieved for the sorting algorithms in [6] are similar to ours. In [7] fault-coverage for AN encoding is reported between 92%–99%, albeit for a different set of benchmarks. The corresponding slow-downs are in the range of $2x$–$64x$. Variants of AN encoding, namely ANB and ANBD encoding, were also studied in [7]. For these encoding schemes slow-downs of up to more than $256x$ were observed, but a fault coverage of well over 99% is consistently achieved across benchmarks.

VI. Related Work

Following the comparative discussion at the end of the previous section we now give a more general account of previous work on software-based error detection techniques.

AN encoding and its variants, ANB and ANBD encoding, were proposed in [5]. ANB extends AN encoding by assigning a static *signature* to each variable. This enables efficient detection of exchanged operands, which may be the result of bit-flips in addresses, as explained in Section V. ANBD encoding also assigns a dynamic *version* to each variable, and thus detects faults that lead to lost updates. The implementations of AN encoding and its variants in [6], [7] were also based on LLVM [9]. In [6] a detailed account of how operations must be modified in order to operate correctly on encoded values is given. ANB and ANBD encoding achieve fault coverage rates of well over 99% but slow-downs may be as bad as several $100x$. The trade-off between fault coverage and performance in AN encoding was analyzed in [8].

Dual modular redundancy (DMR) detects faults by duplicating instructions and comparing results. To facilitate DMR, automated source-to-source transformations were implemented in [12], which requires compiler optimizations to be disabled in order to ensure that the transformations are not undone by optimization passes. EDDI [13] implements DMR at compiler level, with a focus on instruction scheduling to exploit instruction level parallelism. It was also noted in [13] that the order in which instructions are scheduled can affect the efficiency of

detecting faults that lead to invalid control-flow. SWIFT [14] adds control-flow checking to EDDI and also implements simple optimizations at compiler level. ESoftCheck [15] is similar to SWIFT but implements optimizations to remove so-called *non-vital* checks. The fault coverage that is achieved by EDDI is comparable to ours, while SWIFT detects practically all faults. DMR schemes usually lead to slow-downs below $2x$. The advantage of AN encoding over DMR schemes is that permanent hardware faults can also be detected. Moreover, duplication of memory accesses causes issues when DMR schemes are applied on shared memory systems. In AN encoding memory operations are protected without being duplicated.

Δ-encoding [16] merges AN encoding with DMR. Similar ideas were already pursued in [17], [18]. Although the focus of [18] was on fault recovery, it was already acknowledged that scheduling checks close to the uses of values may improve reliability of software-based error detection schemes. Like SWIFT, Δ-encoding also achieves practically full fault coverage. The slow-downs incurred by Δ-encoding are greater than in DMR schemes but generally much lower than for AN encoding. However, Δ-encoding is implemented in [16] as a source-to-source transformation and we believe that it would benefit from improved code generation strategies analogous to our improved checks.

VII. Conclusion and Outlook

Our work has demonstrated that clever implementation decisions can affect instruction scheduling in ways that benefit the quality of software-based error detection. The presented improvements to AN encoding have led to fault coverage of over 98% while reducing the performance overhead by up to 45%. However, slow-downs due to encoding, especially when pointers are encoded, remain large. Our data shows that encoding pointers is crucial to a high level of fault coverage.

In general terms, we have demonstrated that fault coverage can be improved while at the same time lowering the performance overhead. This motivates the development of encoding-specific compiler intrinsics and passes that aid the compiler in generating efficient code that is hardened against hardware faults. Specifically, one could look into ways of giving hints to the compiler that will allow it to reduce the sizes of vulnerable code sequences. Furthermore, understanding the wide variation of speed-ups in Table III might lead to ideas for further improving performance.

Acknowledgments

This work was partially funded by the German Research Council (DFG) through the Collaborative Research Center CRC 912 “Highly-Adaptive Energy-Efficient Computing” (HAEC) and the Cluster of Excellence ‘Center for Advancing Electronics Dresden’ (cfaed). The authors would like to thank Dmitrii Kuvaiskii and Christof Fetzer for various discussions that helped motivate this work.

References

[1] E. B. Nightingale, J. R. Douceur, and V. Orgovan, “Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs,” in *Proceedings of the Sixth Conference on Computer Systems (EuroSys ’11)*. ACM, 2011, pp. 343–356.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM errors in the wild: A large-scale field study,” in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’09)*. ACM, 2009, pp. 193–204.

[3] M. B. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” in *Proceedings of the 49th Annual Design Automation Conference (DAC ’12)*. IEEE, 2012, pp. 1131–1136.

[4] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.

[5] P. Forin, “Vital coded microprocessor principles and applications for various transit systems,” in *Control, Computers, Communications in Transportation: Selected Papers from the IFAC/IFIP/IFORS Symposium, Paris, France*, 1989, pp. 79–84.

[6] C. Fetzer, U. Schiffel, and M. Süßkraut, “AN-encoding compiler: Building safety-critical systems with commodity hardware,” in *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP ’09)*. Springer, 2009, pp. 283–296.

[7] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, “ANB- and ANBDmem-encoding: Detecting hardware errors in software,” in *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP ’10)*. Springer, 2010, pp. 169–182.

[8] N. A. Rink, D. Kuvaiskii, J. Castrillon, and C. Fetzer, “Compiling for resilience: the performance gap,” in *Proceedings of the Mini-Symposium on Energy and Resilience in Parallel Programming (ERPP), Edinburgh, Scotland*, 2015.

[9] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (GCO ’04)*. IEEE, 2004, p. 75.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’05)*. ACM, 2005, pp. 190–200.

[11] G. Paoloni, “How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures,” 2010. [Online]. Available: http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[12] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, “A source-to-source compiler for generating dependable software,” in *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2001, pp. 33–42.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software implemented fault tolerance,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO ’05)*. IEEE, 2005, pp. 243–254.

[15] J. Yu, M. J. Garzarán, and M. Snir, “ESoftCheck: Removal of non-vital checks for fault tolerance,” in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’09)*. IEEE, 2009, pp. 35–46.

[16] D. Kuvaiskii and C. Fetzer, “Δ-encoding: Practical encoded processing,” in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. IEEE, 2015.

[17] N. Oh, S. Mitra, and E. J. McCluskey, “ED4I: Error detection by diverse data and duplicated instructions,” *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, February 2002.

[18] J. Chang, G. A. Reis, and D. I. August, “Automatic instruction-level software-only recovery,” in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*. IEEE, 2006, pp. 83–92.